MICRO-SCHEDULING AND ITS INTERACTION WITH CACHE PARTITIONING

A Thesis
Presented to
The Academic Faculty

by

Dhruv Choudhary

In Partial Fulfillment
of the Requirements for the Degree
Master of Sciences in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2011

MICRO-SCHEDULING AND ITS INTERACTION WITH CACHE PARTITIONING

Approved by:

Professor Sudhakar Yalamanchili, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor George Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Hyesoon Kim
College of Computing
Georgia Institute of Technology

Date Approved: 1 July 2011

To my parents,

Sanjeev and Poonam Choudhary,

PREFACE

This thesis represents a culmination of research that took place over two years from Spring2009 to Spring2011. While the thesis addresses specific problems in building asymmetric multicore architectures, a large part of my time was spent in developing a cycle level simulator that could support a heterogeneous environment consisting of different types of cores. A lot of the understanding and insight presented in the thesis is a direct reflection of the hours of hard work put in designing models that perform similar to commercial processors. The core models developed by me were directly integrated into a larger multicore parallel simulation framework and thus I got exposure to working on many aspects of large core count simulation environments. Apart from my thesis I also worked on a couple of publications with my colleagues Mitchelle Rasquinha and Syed Minhaj Hassan. The first conference publication appeared in International Symposium of Low Power Electronic Design '2010 and was based on evaluation of on chip memory hierarchies constructed from magnetic Random Access Memories (RAM) like Spin Torque Transfer RAM . The second publication is under review at the time of writing this thesis. It highlights the interactions between on chip networks and DRAM systems.

ACKNOWLEDGEMENTS

There are many people who have helped me at various stages during the course of these two years. I would like to express my gratitude to everyone who helped me along the way.

I would like to thank my parents who are responsible for everything I have achieved in my life. They completely supported my decision to spend a year more in school so that I could finish my thesis to my satisfaction

I would also like to thank my advisor, Dr. Yalamanchili for providing me with a platform where I could learn and research on state of the art technology. My colleagues Mitchelle Rasquinha and Nawaf Almoosa have given me invaluable feedback time and again and I have learnt a lot from them during my thesis. I thank all the other colleagues at CASL who I have interacted with technically or otherwise. I would also like to thank Tushar Kumar for helping me think more clearly about the problem I was solving and giving me feedback on my solution quality.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This thesis explores the sources of energy inefficiency in asymmetric multi-core architectures where energy efficiency is measured by the energy-delay squared product. The insights gathered from this study drive the development of optimized thread scheduling and coordinated cache management strategies in an important class of asymmetric shared memory architectures. The proposed techniques are founded on well known mathematical optimization techniques yet are lightweight enough to be implemented in practical systems.

CHAPTER I

INTRODUCTION

While Moore's law continues to produce more cores on a die, dies sizes are not increasing at anywhere near the same rate. Consequently power densities continue to increase while there is increasing pressure on the power budget per core or per tile. These reduced budgets can manifest themselves in a number of different architecture configurations exploiting various forms of core asymmetry. Several recent works([14], [13]) have proposed frequency asymmetric systems where cores are designed to run at different frequencies and applications are mapped to cores according to their compute requirements. Another form of asymmetry, that we evaluate in this thesis is that of using cores custom designed at different energy-delay design points(assuming they operate at the same frequency). This is similar to previous work([5], [8]) where fat *Merom* type cores are coupled with thin *ATOM*-like cores giving us different energy-delay design points in a chip multiprocessor (CMP). Figure 1 illustrates the behavior of the SPEC2006 benchmarks on two different types of cores in terms of a performance energy metric which we define in Chapter III. It illustrates that the energy efficiency across applications is highly variant. This implies that applications have Instruction Level Parallelism(ILP) and Energy per Instruction(EPI) characteristics spread across a wide spectrum.

In this work we i) study the behavior of energy efficiency as the power budget per core/tile decreases and ii) explore approaches to achieve maximum energy efficiency in such an asymmetric environment. Energy efficiency is a growing issue not just for mobile platforms but also for data centers and HPC environments. The two major

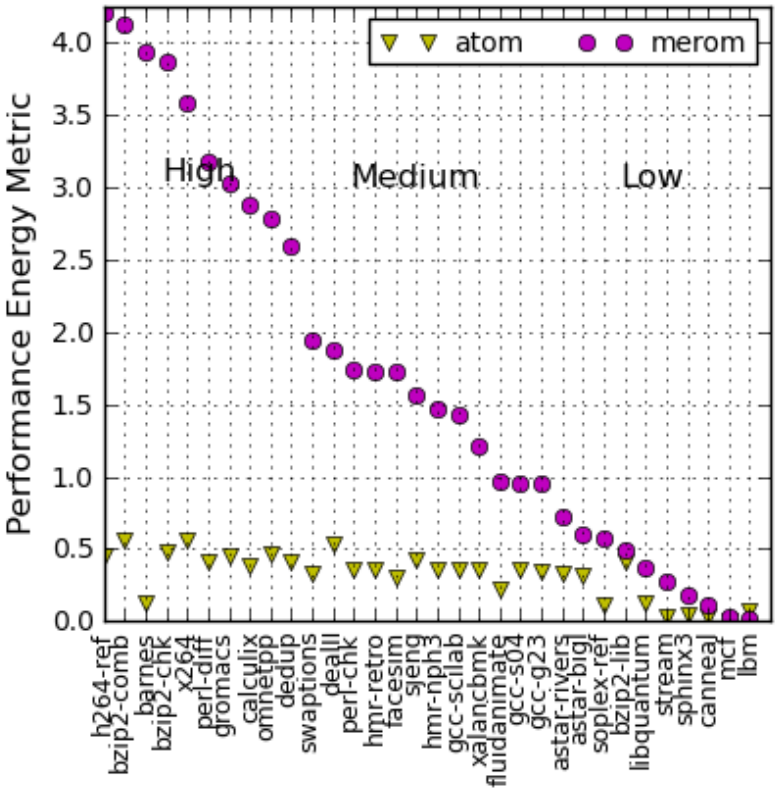

Figure 1: Workload variation on different core types.

energy related costs in a data center are that of the utility kWh charge and the power cooling systems [2]. The reducing power budget per core exacerbates the energy efficiency problem because we need to build simpler cores which consume less power but they service workloads at a lower rate as well. Thus, the whole data center needs to operate for much longer which might increase both the kWh utility cost as well as the energy to cool down the data center.

While asymmetric architectures address the increasing constraint on power, the energy efficiency of these systems may still vary depending on the application characteristics. This is because energy is dependent on the total execution time of a program. Thus although we might use less power on a thin core, an energy efficiency metric such as energy delay squared(ED^2) might increase because the thin core might take much longer to complete the same work compared to a fat core.

This thesis first explores the sources of energy inefficiency in asymmetric multicore architectures where energy efficiency is measured by the energy-delay squared product. The insights gathered from this study drive the development of optimized thread scheduling and coordinated cache management strategies in an important class of asymmetric shared memory architectures. The proposed techniques are founded on well known mathematical optimization techniques yet are lightweight enough to be implemented in practical systems. To summarize, we make the following contributions in this thesis:

- We propose micro-scheduling - a technique to exploit performance and achieve energy efficiency in asymmetric multicore architecture.
- We describe a low complexity optimization framework for the periodic on-line computation of energy efficient thread schedules based on the notion of thread utility - a measure of the threads affinity to a certain core type.
- We highlight how cache partitioning interacts with these thread schedules on an asymmetric substrate and propose a *co-ordinated thread scheduling and cache partitioning scheme*
- We provide a detailed analysis of our approach under increasingly stringent power budgets.

CHAPTER II

ENERGY BEHAVIOR OF APPLICATIONS

This section explores the energy behavior of individual applications on cores of varying complexity. The experiments are carried out in a configuration where there is one traditional out-of-order OOO) core called the fat core and one in-order, two-way superscalar core called the thin core. A thread is executed on one core or the other by a thread scheduler. We first study the behavior of a thread as it moves between the fat core and the thin core. In particular we explore the behavior of a the energy-delayed-squarded metric. All simulation are conducted using a full system x86 multicore simulation infrastructure described in Chapter III.

Figure 1 indicates that applications have computational demands spread across the spectrum. Thus it is not possible to build one core type for all the applications. Ideally each phase of each application should execute on a core defined at a particular design point that suits it, but this is not practical. Moreover applications are not clustered into groups that we can define a core for each cluster. We need to design core types in the middle of the spectrum and try to fit applications to the best possible energy-delay point.

Figure 2 illustrates the variation in ED^2 as an increasing percentage of the work that is executed on a fat core for 6 SPEC 2006 benchmarks. All applications have higher values of ED^2 on the thin core but as an increasing percentage of work is executed on the fat core the ED^2 drops by different amounts for different threads which is indicative of variance in application ILP and EPI properties. The optimization challenge then is to define time varying mappings between these applications and the asymmetric cores to best utilize the system for energy efficiency. For traditional

symmetric multiprocessors this has been a responsibility of the operating system(OS) which is responsible for mapping applications to cores while achieving objectives like fairness and throughput. For asymmetric multiprocessors, an OS scheduling approach sacrifices performance because an OS scheduling quantum is of the order of 100 to 200 msec and applications can have compute and memory phases much smaller than that. Thus the OS scheduler may be unable to match application phases to the core type that suits them. Reducing this quantum is certainly an option but that leads to excessive overhead when the quantum is an order of magnitude less.

We propose a different approach where a hardware level custom circuit or micro controller makes fast scheduling decisions for fast migrations of the order of 5 to 10 msec. We term this *Micro-scheduling*. Migrating a thread at a granularity smaller than the OS quantum has two main advantages. First, for performance, it allows us to be much more responsive to program phases, for example we can run compute phases on fat cores and memory phases on thin cores. Second, due to the decreasing power envelope each technology generation, its is going to get tougher for an OS level scheduler to react to the sudden increases in power of applications. A micro-scheduler on the other hand can be much more responsive to sudden surges of power.

Apart from the obvious difference in scheduling quanta between the OS scheduler and the micro-scheduler, there are other subtle differences between the two. The micro-scheduler needs to be much faster and complex scheduling policies and optimizations are infeasible. Although the micro-scheduler's objective is maximizing performance and energy efficiency, its fairness properties should not contradict OS fairness policies. Micro-scheduler and OS policies should be co-ordinated. There are other implications of using such micro-schedules in a real system. An OS level scheduler interacts very coarsely with the micro-architecture, but a micro-scheduler as the name suggests interacts very closely. Let us take the example of the shared level2 cache. The micro-scheduler needs performance data of applications like Instructions

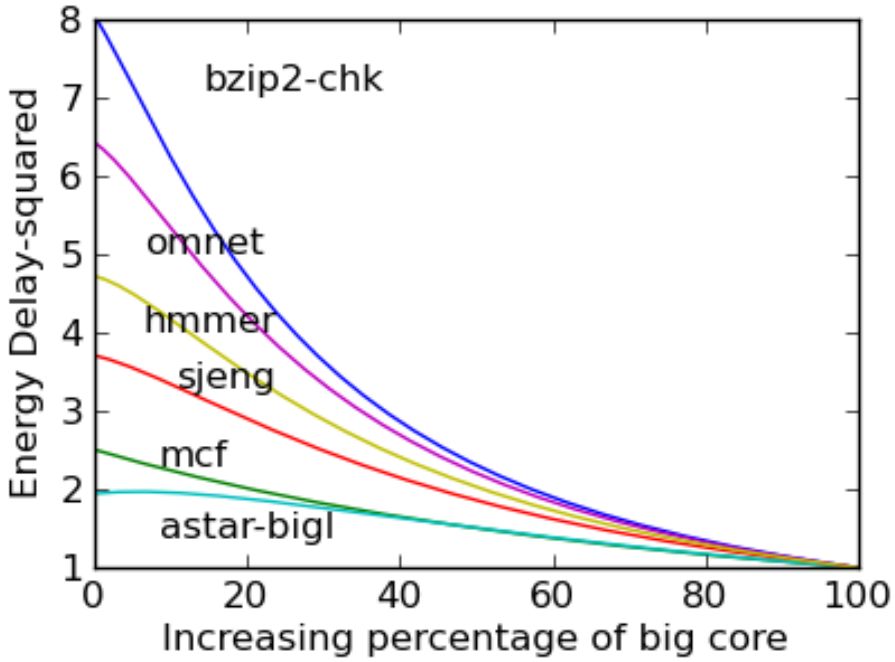

Figure 2: Varying spectrum of slopes indicative of asymmetry in workloads.

per Cycle(IPC) and Energy per Cycle(EPC). These values are collected from a run-time performance monitoring unit. However, the measured values are dependent on the way applications interact in the cache. The inherent asymmetry in the cores creates inherent asymmetry in the rate at which threads demand cache resources. Thus a particular schedule calculated by the micro-scheduler has an impact on the cache behavior which in turn affects the measured IPC and EPC values. This cross interdependence can lead to pathological behaviors where certain high IPC threads dominate the low IPC threads starving them of the fat core and cache space. Conventionally this problem has been solved by cache partitioning schemes like [11], [15] but most of these techniques are oblivious to the thread scheduler and vice versa. The partitioning calculated in one quantum of a micro-schedule might not be optimal for the next schedule and may at worst be disruptive. Thus the cache partitioner can make better decisions if it is aware of the changing micro-schedules.

In this work we use micro-schedules to design a highly energy efficient asymmetric system. We do so by describing a scheduling framework that allocates applications

compute resources according to thread utility - a measure of the benefit of giving an application ILP resources. We describe ways of computing this utility at runtime at minimal cost. The thread utility has a relationship to power. A reducing power budget would require the micro-scheduler to compute utilities that would adhere to the power budget. We describe mechanisms by which the micro-scheduler can adapt to the power budget while achieving the best energy efficiency possible.

We then use this thread utility concept to illustrate how a micro-scheduler should be co-ordinated with shared level 2 cache partitioning. We do so by modifying cache insertion and promotion policies to incorporate the thread utility values in a way that the micro-scheduler and the partitioner share a symbiotic relationship.

2.1 Concept of micro-schedules : Motivation

Figure 2 explores the impact of scheduling a thread on a fat or thin core sharing a level 2 cache. We evaluate the ED^2 metric as a function of the percentage of time the thread executes on the fat core. All applications are executed alone and they have variable increase in ED^2 on the thin core (points on the y-axis) compared to that on the fat core. An application like *astar* has least benefit of being executed on a fat core. In contrast *bzip2* has the highest decrease in ED^2 with the increasing percentage of fat core.

Now consider the case when two applications are executed on a two core system one fat core and one thin core sharing a tile. The performance is evaluated as the weighted ED^2 this is computed by normalizing the ED^2 of a thread to its optimal ED^2 (obtained by executing this thread in isolation with an optimal schedule).The optimal energy efficiency that can be achieved is dictated by the slopes of the curves in Figure 2. Figure 3 plots the weighted ED^2 of four applications pairs that execute a varying percentage of their work on the fat core with respect to each other. The graphs are U-shaped convex curves with the minima in each pair being a different

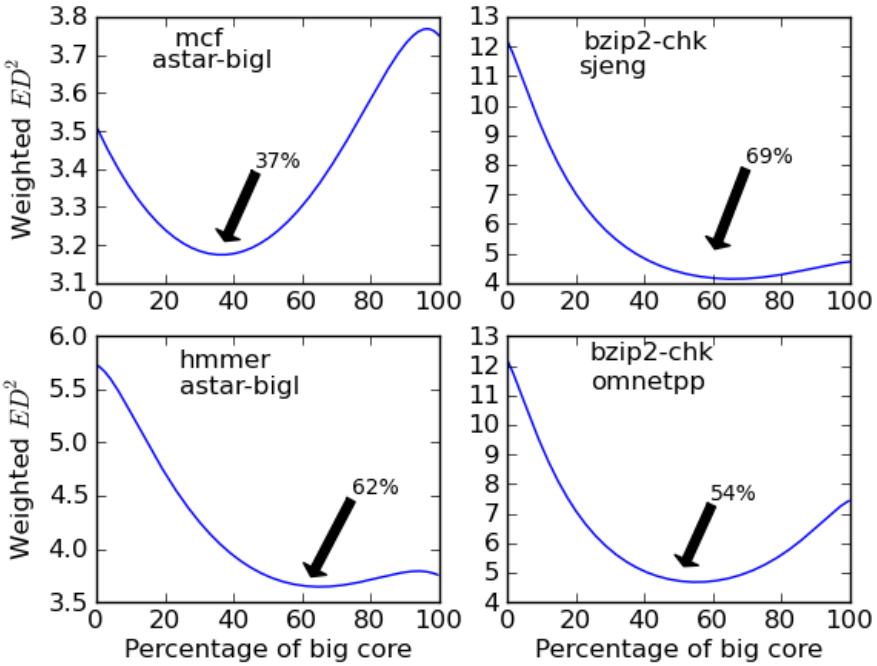

Figure 3: Convex nature of weighted ED^2 function.

point which demonstrates the relative core-bias between the two applications. When executed in isolation applications may achieve optimal energy efficiency on the fat core or the thin core, or a combination of both if it has varying compute phases, but when executed together they need to be given a fat core percentage according to the cost that they expend in ED^2 as they move along the curves in Figure 2. This is because applications have various compute and memory phases and this minima point changes accordingly. The role of a micro-scheduler is to adapt to these changing minima. An OS scheduler due to its larger quanta has low reactivity to these changing minima values. This phenomenon of adapting to frequent changes of application phases is what we term as *dynamic core sharing*. This relative core bias between threads is closely related to our concept of thread utility because it captures the efficiency with which a thread executes on a particular core type.

2.2 Choosing Micro Scheduling Quantum

An advantage of using micro-schedules is its ability to react to sudden power surges. Operationally, when a power surge (for example due to workload changes) occurs threads can be rescheduled to reduce power consumption for example by moving high IPC threads to the thin core. However, before thread migration can have an impact the power may overshoot the budget for some period of time. Figure 4 plots the

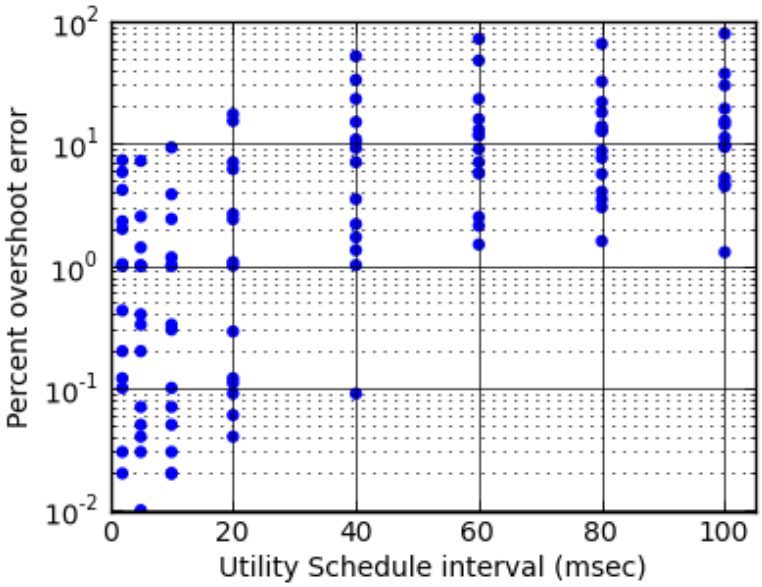

Figure 4: Percentage overshoot error as migration intervals change.

percent overshoot error on log scale for different migration intervals starting from $2msec$ upto OS scheduling quantum of $100msec$. Choosing the interval is a trade off between thread migration costs and the scheduling algorithms ability to adhere to a power budget. Scheduling intervals of 10 to 20 $msec$ show lower power budget overshoot error than higher intervals, which indicates that the micro-scheduling quanta must be at least an order of magnitude smaller than the OS scheduling quanta. We choose $10msec$ (roughly 30 million cycles at 3 GHz) as the scheduling interval because at $10msec$ and below our experiments indicate that we can compute schedules that adhere to within 5% of the power budget. The few outliers are due to workloads which are comprised of only high IPC threads. As we show later, for these kind of

workloads thread scheduling as a power management technique is not very effective. All results presented hereafter will assume a scheduling interval of 30 million cycles unless otherwise specified.

2.3 Primary Concept

Our study of the energy behavior of applications described in the preceding sections leads to the following intuition - a thread should be allocated a share of the fat cores in proportion to its demand for work, where work may be measured by IPC for example. The demand for computation made by a thread is captured in the concept of thread utility. Scheduling policies are driven by thread utility.

CHAPTER III

ASYMMETRIC ARCHITECTURE CONFIGURATION

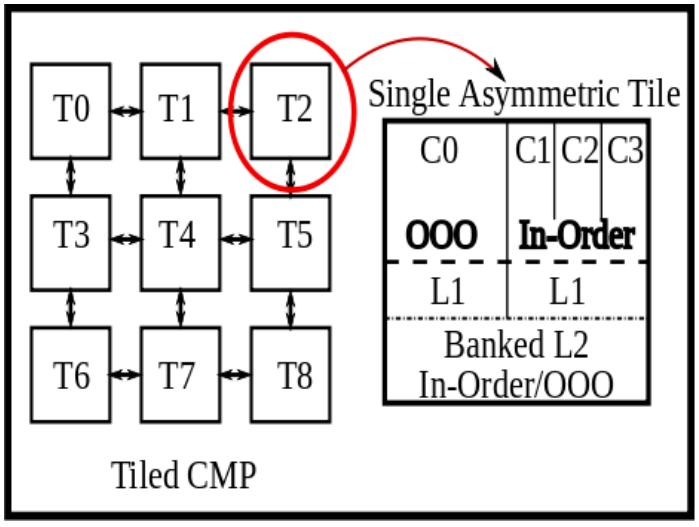

Figure 5: Architecture of Asymmetric Configuration.

Asymmetric architectures can be constructed in many different system configurations. Although the proposed policies and techniques can be extended to all configurations, we choose one that we think has several useful properties. Figure 5 shows the system architecture where each tile of the CMP is assembled with a fat core and few thin cores. A level 2 (L2) cache is banked across the whole chip and all cores share the cache. This formation reduces migration costs and also reduces the cost of verification and testing because the tiles are symmetric. The thin cores in a tile share the level 1 (L1) cache. Although we use cores that are single threaded, most commercial inorder cores are 2 to 4 way multi threaded eg. Niagara and ATOM. Thus it is not an unreasonable assumption for 2 to 4 inorder cores to share a single L1 cache.

Table 1 provides the machine configuration of fat, thin cores and the caches. We

Table 1: Table of system configuration values

	Merom	Atom
Branch Predictor	TAGE(4KB), 2K-entry/4-way BTB RAS-16	2bC(1KB), 1K-entry BTB RAS-4
DTLB	16 entries/4-way	4 entries/4-way
ROB, RS, LDQ, STQ	96, 32, 32, 20 ByteQ-16	ByteQ-4
Decode/Issue	uOP Queue 24, Dec 4-1-1-1 w/ fusion	uOP Queue 8, Dec 4-1
Pipeline Units	3 INT-FUs, 2 FP-FUs, 3 MEM units	2-way superscalar 2 INT-FUs, 2 FP-FUs
Pipeline Latency	3-cycle add, 5-cycle ,mult 24-cycle div	3-cycle add, 8-cycle mult, 40-cycle div
Instruction L1	(64KB) 4-way, 64 byte lines	(64KB) 4-way, 64 byte lines
Data L1 Cache	(64KB) 4-way, 64 byte lines, 16 MSHR's	(64KB) 4-way,64 byte lines, 32-MSHR
L2 Cache	1MB per bank/tile,2-banks,32-way,64 byte lines, 32-MSHR	
Router	5-stage DOR, Round Robin SA	
MC Policies	FR-FCFS, Page Interleaving, Open Page	
3 Ghz, MOESI Coherence		

use a cycle accurate x86 simulator (Zesto [9]) which was modified to run with a QEMU front-end that emulates a Linux image. The backend was interfaced with a MOESI coherent cache-network simulator. Zesto handles micro-ops in the pipeline instead of macro-ops and thus whenever we refer to committed instructions per cycle, we are referring to the number of micro-ops committed rather than the number of macro-ops. We use MCPAT [7] for energy and power modeling. We run most simulations for 2 billion instructions to assess the performance of micro-scheduling and its interaction with the OS scheduler.

We choose 23 multiprogram workloads from SPEC2006 suite(Figure 1). We always execute as many threads as the number of cores. The scheduling of a larger number of threads than cores is handled by the OS scheduler. Figure 1 classifies the workloads into high medium and low performance. From these we form 16 combinations of workloads with high, medium and low inter-application variance. Workloads are

chosen to stress the scheduling and cache partitioner. We also choose a workload with all high IPC threads which illustrates situations where the proposed techniques fail.

Table 2: Table of workload mixes

WL0	bzip2-chk sphinx3 soplex-ref lbm
WL1	calculix sjeng astar-bigl mcf
WL2	h264 libq gcc-s04 mcf
WL3	gromacs bzip2-comb soplex-ref bzip2-lib
WL4	gromacs calculix xalancbmk gcc-g23
WL5	h264 h264 h264 h264
WL6	h264 hmmer-ret bzip2-comb sphinx3
WL7	h264-ref mcf bzip2-lib astar-riv
WL8	h264 perl-chk astar-rivers lbm
WL9	libq sjeng gcc-s04 soplex-ref
WL10	omnetpp xalancbmk hmmer-nph3 lbm
WL11	perl-diff dealII gcc-g23 bzip2-lib
WL12	perl-diff gromacs xalancbmkastar-riv
WL13	perl-diff omnetpp hmmer-ret perl-chk

3.1 Performance metrics

To quantify throughput we use a metric that can capture the amount of work performed but without penalizing the low IPC threads. Thus we choose *weighted IPC* and *harmonic IPC* which are defined as

$$wIPC = \sum_{i=1}^{n} (IPC_{shared})_i / (IPC_{alone})_i \tag{1}$$

$$hIPC = n / \sum_{i=1}^{n} (IPC_{alone})_i / (IPC_{shared})_i \tag{2}$$

Similarly when we define energy efficiency as a product of energy and delay, we want to capture the best possible value without penalizing threads with the largest ED^2 value. The most commonly used energy efficiency metrics are Energy Delay

product(ED) or the Energy Delay-Squared product(ED^2). For a given thread i

$$ED_i = (e_i * (M_i)^2)/(IPC_i)^2 \tag{3}$$

Table 3 provides the nomenclature. As we saw in Figure 2, a given application would get the optimal ED^2 when run on the fat core or thin core or a combination of the two(depending on compute and memory phases). Thus we need to weigh the energy delay metric according to these optimal values. The square in the term shows that

Table 3: Metric parameters

e_i	energy per cycle for thread i
IPC_i	instructions per cycle for thread i
M_i	work to be done in instructions for thread i

energy is a function of the delay as well. Thus our weighted performance energy metric ($wPEM$) and harmonic performance energy metric ($hPEM$) analogous to ED for n threads is

$$wPEM = \sum_{i=1}^{n} \frac{IPC_i^2/e_i}{(IPC_i^2/e_i)_{opt}} \tag{4}$$

$$hPEM = n/\sum_{i=1}^{n} \frac{(IPC_i^2)/e_i)_{opt}}{IPC_i^2/e_i} \tag{5}$$

If we instead optimize for the energy delay-squared (ED^2) product all the quadratic terms become cubic. Our baseline for comparison is round robin where all threads operate at a point on the curve where they get equal share of the fat core.

3.2 Operational Model

With respect to Figure 5, the system comprises of two levels of schedulers - the OS scheduler that runs at a higher temporal granularity and the hardware scheduler or micro-scheduler that runs at much finer temporal granularity. The OS scheduler makes scheduling decisions across the chip-multiprocessor (CMP), but the CMP may consist of many micro-schedulers working independently of each other. Each micro-scheduler may either consist of a single tile or multiple tiles. These tiles comprise

the region where thread migrations are confined to and is called the *domain* of the micro-scheduler. The organization of tiles is motivated by an intuition that suggests that to achieve energy efficiency we must i) make cores of varying energy efficiency available to a thread, and ii) we must reduce the cost of thread migration. The preceding organization provides opportunities for both by have multiple core types that share a level of the memory hierarchy, in this case the L2 cache.

The OS scheduler may be unaware of the asymmetric composition of cores in a tile and schedules threads to micro-scheduler domains. The OS scheduler also provides a power budget to each micro-scheduler domain. Micro schedulers operate independently of each other. Each micro-scheduler assigns to each application (a single thread) in its domain a value called the *thread utility* which indicates the utility of executing an application on a certain core type. A thread's utility changes over time with its behavior as described in Chapter IV. In addition to making scheduling decisions, thread utility values are also used to coordinate the sharing of the L2 cache. The level 2 cache is banked and shared across all the micro-scheduler domains. The cache partitioner is unaware of the domain to which a thread belongs. Instead it is only concerned with the utility of a thread in making partitioning decisions.

Operationally the OS scheduling interval is divided into micro-scheduling intervals of duration T cycles. Every T cycles the micro-scheduler is invoked, optimized scheduling decisions are made, and threads migrated between the fat cores and thin cores in the domain.

CHAPTER IV

THREAD UTILITY

Thread scheduling is often formulated as a graph assignment problem of mapping threads to cores and has been studied in extensive detail at the OS level([5], [8]). We have already shown the advantages of dynamically sharing cores between applications at a finer granularity than the OS scheduling granularity. The problem with graph assignment scheduling at the micro-scheduling level is that it burdens the scheduler with high complexity assignment computation. This formulation makes it difficult to define a system wide integrated cost in terms of the dynamic core sharing we described in Chapter II.

We instead formulate a general framework that allows us to define a number of different cost functions such as throughput, uniform performance or energy efficiency. To accomplish this we first define the concept of *thread utility*.

In general, the thread utility is a measure of the utility of executing a thread on a specific type of core. It attempts to capture the benefit, in terms of energy efficiency, of executing a thread on a specific core. In this thesis, the thread utility is represented as a number of cycles (or seconds) a thread executes on a particular core type in a micro scheduling interval. This is different from a graph assignment formulation because within every micro-scheduling interval, a thread receives a share of a core proportional to its utility on that core. This enables fast scheduling decisions and enables us to derive cost functions in terms of notions of utility.

4.1 System Model Formulation and Assumptions

One can envision a number of core types designed to occupy various points that trade-off energy efficiency and performance. While, in the near future we probably do not

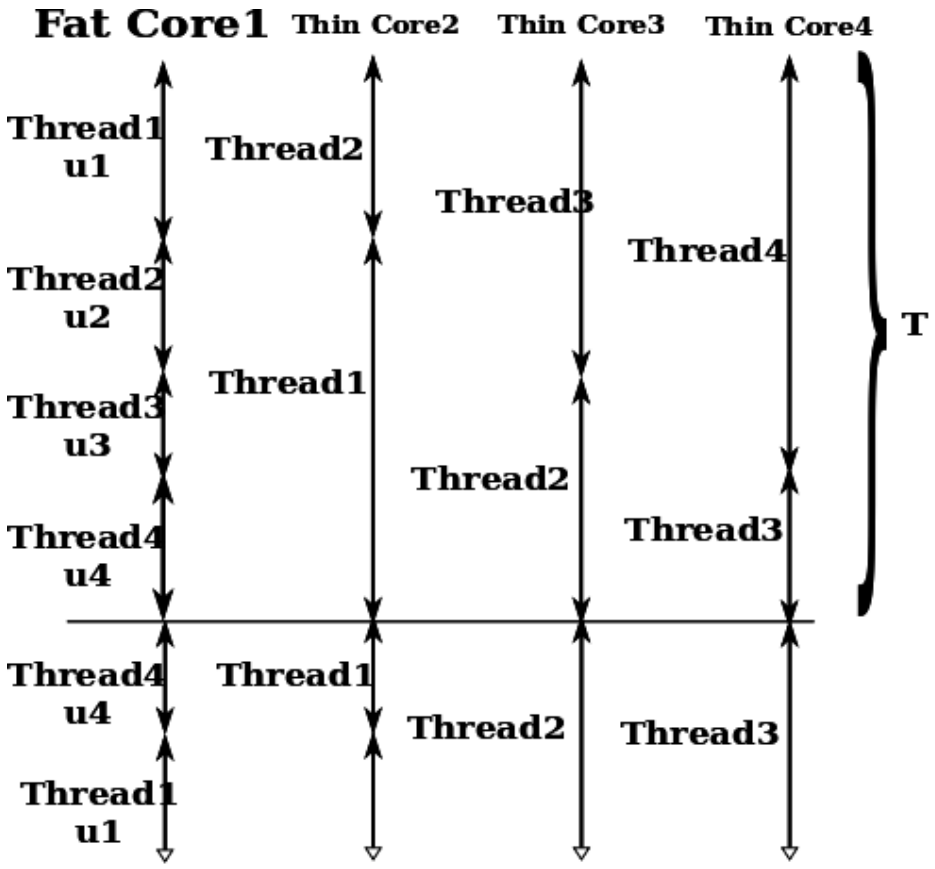

Figure 6: Sequence of thread migrations inside the scheduling quantum 'T'

expect to see more than 2-3 core types in commercial processors, the formulation of the micro-scheduler is sufficiently general. We present it in its most general form with n cores and n threads. Let us consider we have p different core types forming p levels of asymmetry. We refer to a fat core like Merom as the highest level of asymmetry and a thin core like Atom as the lowest level of asymmetry. Every T cycles the micro-scheduler is invoked and it produces a sequence of schedules where each thread spends at least some time on each core of the higher $p-1$ levels of asymmetry. The vector U_i represents the time $thread_i$ spends on each core. Thus $U_i = \begin{pmatrix} u_{1i} \\ u_{2i} \\ \cdots \\ u_{mi} \end{pmatrix}$ where m is the number of cores in the higher $p-1$ levels of asymmetry. Thus we now have a $m \times n$ matrix of times spent in cycles by each thread on each core (in every scheduling interval). The reason we can ignore the times spent on the cores of the lowest level of asymmetry is that it is implicitly captured in the time not spent on the higher $p-1$ levels. This is an important observation because we have many more thin cores than the fat cores. Let us call this U matrix a *Utility Matrix*. Each element of this matrix defines the utility of running a certain thread on a certain core. This is what we refer to as *thread utility*. The total time spent by all the threads on a core, should add upto T. Thus the first set of constraints that arise from this formulation are, the linear equations formed by summing the rows of U.

$$\sum_{i=1}^{n} u_{ji} = T \text{ for all j} = 1 \text{ to m.} \tag{6}$$

The second constraint is that due to the power envelope.

$$\sum_{j=1}^{m}\sum_{i=1}^{n} e_{ji} * u_{ji} + \sum_{i=1}^{n} e_{li} * (T - \sum_{j=1}^{m} u_{ji}) <= P_{budget} * T \tag{7}$$

where e_{ji} is the energy per cycle of thread i on core j. There is an extra term in the power budget to account for the power consumed by the lowest level of asymmetry

and e_{li} is the average energy per cycle of thread i on the lowest level of asymmetry. We now apply this generic formulation to our system where we have only two levels of asymmetry. The utility matrix will consist of the time spent on each fat core. As for the constraints, we have as many linear constraints as the number of fat cores. Thus if we have a single fat core the only linear constraint apart from the power constraint is, $u_1 + u_2 + ... + u_n = T$. where u_i is the time spent on the fat core by thread i. For the majority of the paper we evaluate the configuration with the single fat core and multiple thin cores for simplicity and ease of explanation. In the results section we show that all results are equally applicable to multiple number of fat cores as well. Figure 6 shows the sequence of migrations in a system with single fat core and three thin cores. In this figure u_i represents the time spent on the fat core and implicitly the combined time spent on all the thin cores is $T - u_i$. The purpose of defining a U matrix is to formulate an energy efficiency objective function as described in the next section.

4.2 Cost-to-go ED^2 function

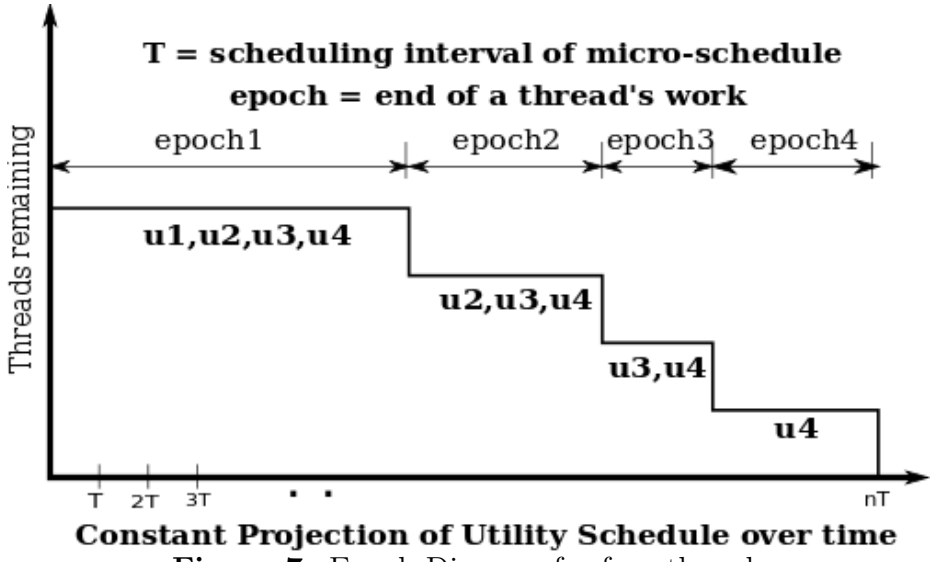

Figure 7: Epoch Diagram for four threads.

The ED^2 metric is computed over the entire program execution while power is an

instantaneous rate. Therefore, if our objective is to minimize ED^2, our solution technique must implicitly or explicitly estimate the remaining execution time or amount of work (e.g., number of instructions) remaining for an application. After every T interval the micro-scheduler is invoked and it estimates a combined cost of how much work is left and then produces a schedule to minimize that cost under the system constraints.

To estimate the future cost we assume that the IPC and energy profiles of thread are constant till they complete, and the share of the fat core (according to their utility) given to them per interval is the same throughout, they will end at different times. The cost function should incorporate the time they spend and the energy they take till they complete. Figure 7 shows a cost diagram where each epoch is marked by the end of a threads execution. Here u_i is the time spent by $thread_i$ on the fat core. The threads migrate according to the micro-schedule every T interval as shown in Figure 6. Even after a thread completes execution, some other thread may be scheduled by the OS to replace it, so our approximation is not unreasonable in assuming that the u_i's of the remaining threads don't change considerably. Ee_i and De_i are the energy and delay of the thread that finishes in $epoch_i$. Let us assume that amount of work left in each thread is M_i at any instant.

Let Xe_i be the work done in a scheduling interval T. Thus,

$$Xe_i = IPCe_{fat-i} * ue_i + IPCe_{thin-i} * (T - ue_i) \tag{8}$$

where $IPCe_{fat-i}$, $IPCe_{thin-i}$ are the measured *instructions per cycle* on the fat core and thin core for thread finishing in epoch i. Let Ye_i be the energy spent in a scheduling interval T. Thus,

$$Ye_i = EPCe_{fat-i} * ue_i + EPCe_{thin-i} * (T - ue_i) \tag{9}$$

where $EPCe_{fat-i}$, $EPCe_{thin-i}$ are the estimated *energy per cycles* on the fat and thin

core. Our assumption of uniform projection of instantaneous cost is not an unreasonable one as long as the IPC's change gradually and we re-evaluate this function regularly to update the cost with instantaneous IPC and EPC values. In practice this constant projection works quite well. But the drawback is that it works only if it is evaluated regularly, so that the error in the model does not lead us to a non-optimal path. Thus the delay and energy of a thread in an epoch is

$$De_i = (Me_i * T)/Xe_i, \tag{10}$$

$$Ee_i = (Ye_i * De_i)/T \tag{11}$$

Thus our complete cost function is

$$min_U L(U) = \sum_{i=1}^{n} (Ee_i * De_i^2) \tag{12}$$

In practice this formulation is incomplete because threads with very high ED^2 would be penalized. Thus we need to weigh the individual terms appropriately to formulate the final function. We describe in Chapter V how we can approximate these weights at runtime. It should also be noted that the formulation is minimizing the inverse of the harmonic Performance Energy Metric(*hPEM*). We also formulated it in the weighted form, but the results were not significantly different, thus we omit them for brevity. The constraint due to the Power envelope.

$$\sum_{i=1}^{n} EPCe_{fat-i} * ue_{fat-i} + EPCe_{thin-i} * (T - ue_{fat-i}) <= P_{budget} * T \tag{13}$$

This is a highly complex non-linear function. However, we saw in Chapter II that it has a convex form. We first find the minima of this function assuming we have infinite compute resources to see what is the best we can achieve. To do this we use COBYLA(Constrained optimization by Linear Approximation) [10] algorithm, because of our intuition that the function has a convex form. We empirically observed that this algorithm converges in less than 100 iterations most of the time. We use

the NLopt [4] numerical Optimization library for the same.

In real systems it is unreasonable to expect that we know how much work (Me_i value) is left in a thread. But the advantage we have is that the micro-scheduler lives in the framework of the OS scheduler and thus all its policies work in the bounds of the OS scheduling quantum. Thus we need some notion of how much work can be done in an OS quantum. In our function we use the amount of work the thread would have done if run on the fat core for the whole OS quantum. To calculate this we multiply the average IPC of the thread i in the previous quantum to the OS scheduling interval. Alternatively these Me_i values can factor in things like OS priority levels and quality of service.

Similarly we can define other cost functions as well. For eg: to get predictability in performance we can force the threads to finish close to each other by defining a cost function of squared delay differences between threads. For throughput our objective function can be defined as,

$$min_U T(U) = \sum_{i=1}^{n} (De_i) \tag{14}$$

Let us look at some of the things we need in this model. We need the thread performance in the terms of IPC numbers which we get from performance counters. To estimate the energy we build a linear regression model of the different counters like load queues, reorder buffers, execution pipes, cache misses. The energy of the level2 cache is not significant and so we don't include it in the model but we include it when reporting results.

Given the complexity of this function it is quite clear that such a function is not feasible to implement in hardware and be re-invoked frequently. We use the intuition gathered in analyzing this function to define a weighted quadratic approximation that we describe next.

4.3 Weighted Quadratic approximation

Weighted quadratic functions like the DeJong function have the nice property that their minima can be easily calculated by using the Lagrange variable and the constraints. In our case we have one equality constraint and one inequality constraint of power. Thus if we can approximate the above function by a DeJong function, we can afford recomputation every T cycles. The generic form of the Weighted Quadratic function is

$$L(U) = \sum_{k=1}^{n} a_k * (u_k)^2 \tag{15}$$

In our system u_k represents the thread utility of thread k on the fat core. Thus if we have a constraint $g(U) = 0$ then we can find the minima by solving the derivate of the Lagrangian in U,

$$\frac{d}{du}\left(L(U) + \lambda g(U)\right) = 0.$$

In our formulation, there will be a u_k term for each thread on each fat core that represents the threads utility on that core. To choose the coefficients to the function we use the notion of relative benefit of running a thread on a core with respect to the core from the immediate lower level of asymmetry. This is because it is not only the performance on the fat cores that differentiates threads, it is also their performance on the thin cores that matters.

$$a_i = (EPC_{fat_j - i}/IPC_{fat_j - i})^{\gamma 1}/(EPC_{thin-i}/IPC_{thin-i})^{\gamma 2} \tag{16}$$

where $EPC_{fat_j - i}$ and $IPC_{fat_j - i}$ are the measured EPC and IPC of thread i on fat core j. EPC_{thin-i} and IPC_{thin-i} are the average EPC and IPC values collected on the thin cores. Thus if a core has much better performance per joule on a more powerful core, then its coefficient weight is lower and it consequently gets a higher u_i and hence higher performance. Our constraints are still the same as the complex function. Similarly for throughput the coefficient weights can be defined as,

$$a_i = (1/IPC_{fat_j - i})^{\gamma 1}/(1/IPC_{thin-i})^{\gamma 2} \tag{17}$$

Again in our single fat core configuration the coefficients are,

$$a_i = (EPC_{fat-i}/IPC_{fat-i})^{\gamma 1}/(EPC_{thin-i}/IPC_{thin-i})^{\gamma 2} \quad (18)$$

From hereon we refer to this scheme as *Utility Scheduling* (USCHED). We now discuss the significance of the coefficient parameters in detail.

4.3.1 $\gamma 1$ and $\gamma 2$ Values

There are two potential benefits of the gamma values. First, changing the gamma values allows us to shift the minima of the Dejong function to move it closer to the actual minima. Second, if we look at the above equation without the γ parameters, it gives us a set of u_i values for each thread. But we need to solve this equation with an inequality constraint of power. The γ values have a direct correlation to the Power budget we need to adhere to. Thus this gives a very convenient tunable parameter that can modulate our thread schedules in terms of the Power budget constraints. This turns out very useful because inequality constraints are in general very difficult to solve for and given our constraints a close approximation is good enough. The γ values also help us trade in how much importance we give to high IPC threads with respect to low IPC threads. This is important because it helps us trade in fairness against importance for higher IPC threads. If we give equal weights to high IPC threads and low IPC threads(ie. $\gamma 1 = 1$, $\gamma 2 = 1$), we get fairness but compromise on throughput or ED^2. As we increase $\gamma 1$ we trade in fairness for better throughput or ED^2. Each set of applications has a different energy efficiency sweet spot for values of $\gamma 1$ and $\gamma 2$ as we show in Chapter V.

4.3.2 Adding factor for work left

As we saw above we need to weigh applications according to the amount of work left in a particular thread. So we modify our weighted quadratic function to add a term

for the work left in the thread M_i.

$$L(U) = \sum_{i=1}^{n} (a_i * (u_i)^2 * (1/(M_i)) \tag{19}$$

CHAPTER V

PERFORMANCE EVALUATION OF THREAD SCHEDULES

In this section we present the performance evaluation of using USCHED. Figure 8 shows the $wPEM$ metric, for our chosen set of workloads. We show two USCHED configurations with $\gamma 1$ values of 2 and 3, hereby referred to as $dj21$ and $dj31$. The choice of these values is explained in detail in Chapter VII. The power budget is unconstrained for this plot. For COBYLA we used a constraint tolerance level of $1e-8$, and a relative tolerance for stopping criteria of $1e-4$. The starting condition for the algorithm is set to equal u values. To convert this to weighted ED^2 form, we need to normalize the individual values with estimates of their optimal ED^2. To calculate the weights at runtime, we average the IPC's on the fat core in the last five intervals and extrapolate it to estimate the performance of the threads when executed on the fat core only.

As we can see COBYLA is very agressive in finding the minima. On average

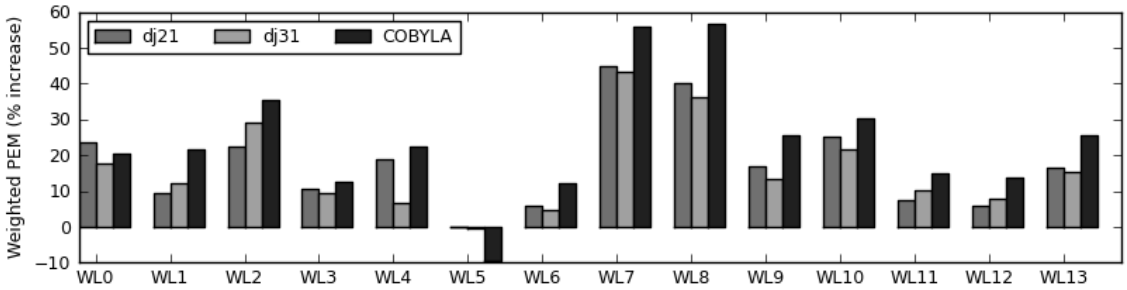

Figure 8: Weighted PEM for Quad core workloads when optimized for ED^2

its $wPEM$ is better than round robin by 22%. The dejong functions are not very aggresive and $dj21$ and $dj31$ give mean improvement of 16% and 14% respectively.

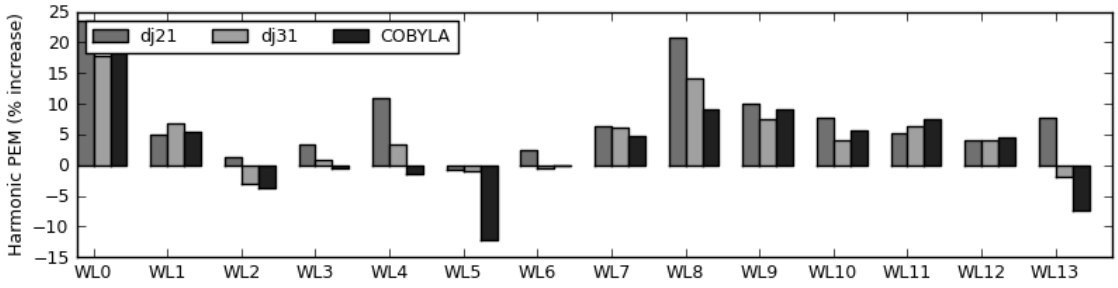

Figure 9: Harmonic PEM for Quad core workloads when optimized for ED^2

Workloads like WL1 and WL2 show more improvement with $dj31$, while others fare better with $dj21$. This is because the Dejong function tries to approximate the minima and for different workloads the minima lies at a different point. Thus gamma parameters play a role in how close the Dejong minima is to the actual minima. Figure 9 shows the $hPEM$ metric for the same configurations. Figure 10 shows that we do not hurt weighted throughput when we optimize for weighted ED^2.

Although the weighted improvements are significant, all minima finding techniques can give unfair advantage to certain applications in trying to find their minima thus hurting the fairness metric. There are two major reasons for that. First, our objective function has a squared delay term due to the ED^2 metric. Thus the algorithm is bound to favor the high IPC threads. The algorithm is very sensitive to even small differences in IPC between threads. The extreme case of that is shown in $WL5$ which is a quad workload of all four *h264* applications. COBYLA gives a negative $hPEM$ for WL5. A small difference in the way the four threads run, which might be because of the order in which they run or cache interference, can lead the algorithm to find a minima which is artificially created by the system interference.

Second, one of the artifacts of using an asymmetric system of cores is that it creates an asymmetry in resource demand, which creates asymmetry in resource utilization leading to more change in demand. This cycle of interaction between applications and their utility schedules can lead to a situation where shared resources like caches can

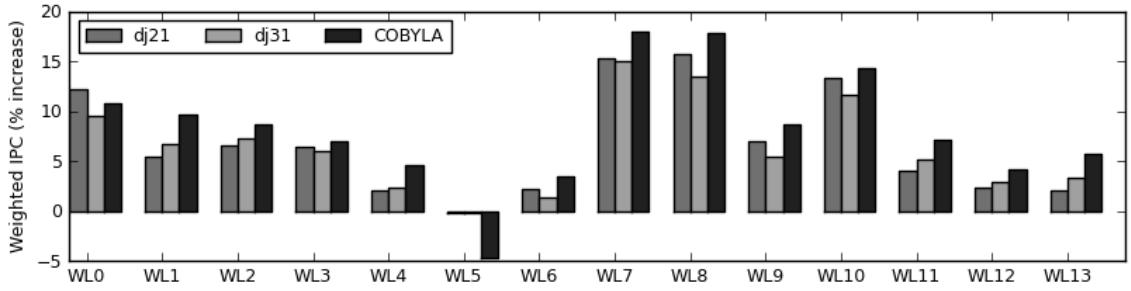

Figure 10: Weighted Throughput for Quad core workloads when optimized for ED^2

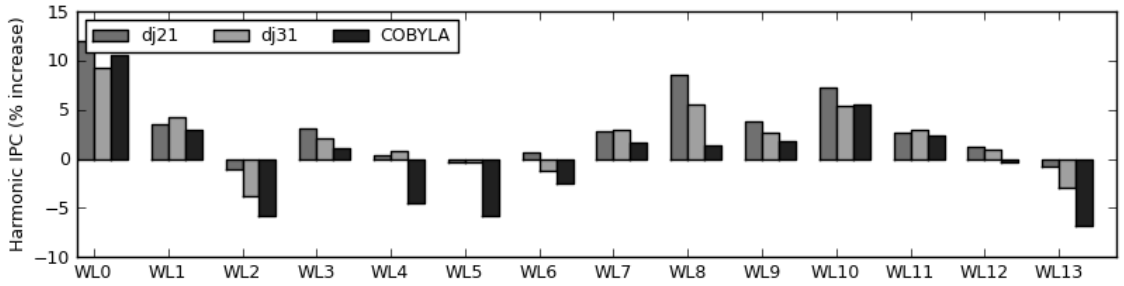

Figure 11: Harmonic Throughput for Quad core workloads when optimized for ED^2

cause pathological interference between threads leading to incorrect calculations of utility schedules. We outline two scenarios that illustrate how this skewed capability of cores may affect the micro-scheduler adversely.

Consider a case with two threads A (high IPC) and B (low IPC). Let us assume A has a high miss rate and thus its demand rate to an L2 is high. Thread B has a reduced demand rate because it runs on the thin core for a larger part of its execution. Thus ThreadA ends up receiving a larger share of the L2 cache thereby reducing the IPC of ThreadB further. An algorithm like COBYLA reacts to this by giving ThreadA more share of the fat core, thereby starving ThreadB even further. Thus we can see that $hPEM$ for COBYLA is severely degraded. USCHED on the other hand is slightly more robust because, a reduction in IPC of ThreadB reduces the weight of ThreadB thereby increasing its share of the fat core. Thus USCHED has slightly better $hPEM$ performance. However, the underlying inefficiency still exists because giving ThreadB a little more share of the fat core will reduce the performance of the

ThreadA. ThreadB on the other hand does not benefit much by the small increase in fat core utility because it still suffers from the interference with ThreadA.

Consider a second case with two threads C (moderate IPC) and D (moderate IPC). Let D be an application that shows thrashing behavior in the cache similar to the ones shown in [11], [3]. In this case both threads will start with relatively equal share of the fat core, but Thread C's share of the fat core is reduced over time due to the interference due to the thrashing in the cache. The micro scheduler is unaware of these interactions and makes decisions based on measured IPC values which leads it to make sub-optimal decisions. We address these inefficiencies in the next section which motivates our co-ordinated cache-partitioning scheme.

CHAPTER VI

CO-ORDINATED CACHE PARTITIONING

Cache partitioning in symmetric systems is based on the assumption that the rate at which threads inject requests into the memory system is only dependent on the application characteristics and independent of the core since all cores are identical. That assumption is no longer valid in an asymmetric architecture because application demand is now also dependent on the core capability. A thread's demand for cache resources depends on the core on which it is executing. The high performance core can issue memory requests significantly faster than the in-order cores. Consequently, the footprint of the fat core in the cache grows as it aggressively fills up the cache. This increases the interference with the thin core reducing its IPC and affecting its utility based schedule. This cycle of interaction continues possibly significantly degrading performance. This observation is particularly important in the context of micro-scheduling since the the IPC of a thread is used to it compute the utility values allocated to different threads. Thus, in general we need a scheme wherein the cache partitioning can be coordinated with the thread scheduling to avoid negative reinforcement between core usage and cache effects.

To this effect we propose a very simple prioritization scheme in the cache. The principle is to give higher priority to threads with low thread utilities (computed by the micro-scheduler). This higher priority is used to influence the cache insertion policy. This is an adaptation of the insertion policy suggested in [15]. We hypothesize that the thread utilities have an inverse relationship with the insertion priority in the cache. A thread with a normalized utility u is inserted at a position u from the top of the replacement stack. We always evict the least recently used line at the bottom

of the replacement stack. Whenever a line is hit, it is promoted to the most recently used position similar to LRU policy.

CHAPTER VII

RESULTS

7.1 Co-ordinated Utility Scheduling and Cache Partitioning

For all results we present the percentage increase over the round robin thread scheduling policy with the unmanaged shared cache (LRU replacement). We do not present any more results with the COBYLA algorithm and all results use the USCHED algorithm. Figure 12 gives the $wPEM$ comparison between USCHED with and without the co-ordinated cache partitioning scheme. The cache partitioning increase the benefit of USCHED from 16% to 22% on average. Figure 13 shows that it increases the $hPEM$ from 5% to 9%. This was expected because low IPC threads now get priority in the cache and thus the amount of work done by them on the thin cores increases which allows USCHED to give a higher share of the fat core to the high IPC threads. Thus the system ends up improving the fairness metric.

Consider the case of WL3. WL3 includes libquantum which can have phases of moderate and low IPC, but its miss rate is generally high. Thus it starves a low IPC application like mcf. The utility scheduler is unaware of this interaction. However, by using the values of thread utility in the cache, mcf has higher priority than libquantum. This increases the work done by mcf on the thin core which leads to better performance and fairness. Libquantum on the other hand is fairly unaffected by the reduced cache priority.

The other extreme case is that of WL12. Here the cache-partitioning using thread utility has no benefit. Here perl has the maximum thread utility and astar has the least. But the IPC of astar is unaffected by the cache partitioning because astar does

not benefit from increased cache space. However the IPC of perl decreases which hurts $wPEM$ slightly.

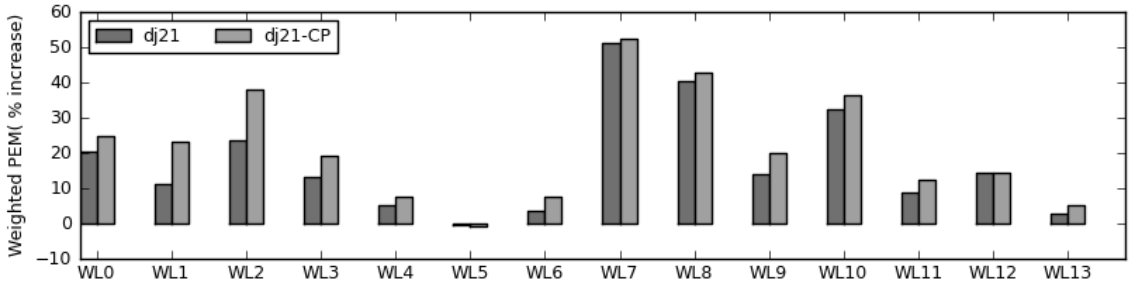

Figure 12: Weighted PEM for Quad core workloads with and without cache partitioning

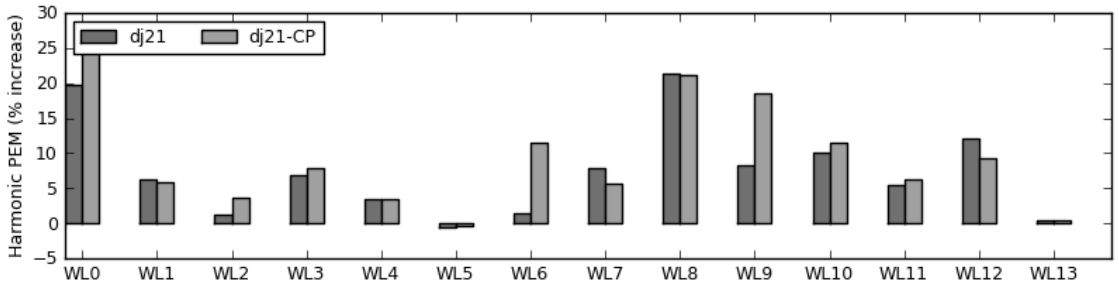

Figure 13: Harmonic PEM for Quad core workloads with and without cache partitioning

7.2 Gamma values and their effect

Choosing the correct values of $\gamma 1$ and $\gamma 2$ is very important to the efficiency of utility scheduling and adhering to the power budget. Figure 14 shows a wire frame plot of gamma values and its effect on weighted ED^2 for a representative workload(WL8). $\gamma 2$ is varied in steps of 0.1 from 0.5 to 1.5. $\gamma 2$ is varied in steps of 0.2 from 1 to 3. Each pair of gamma values defines a unique DeJong function. For this workload the maximum $wPEM$ is given by the point ($\gamma 1 = 2.6$, $\gamma 2 = 0.9$). Other workloads might find a maxima at some other point. We analyzed such plots for all applications and concluded that most maximas fall around $\gamma 1 = 2 - 3$ and $\gamma 2 = 0.8 - 1.2$. This is why we present results for two configurations of ($\gamma 1 = 2$, $\gamma 2 = 1$) and ($\gamma 1 = 3$, $\gamma 2 = 1$).

Figure 15 shows a wireframe plot for average power consumed by all the cores with the γ values. Given a power budget only certain points on the wireframe can satisfy it. To adhere to the budget our Utility Scheduler follows an iterative algorithm to change $\gamma 1$ values. We keep the $\gamma 2$ value constant at 1. The starting condition is always ($\gamma 1 = 2$, $\gamma 2 = 1$). We increase the $\gamma 1$ step size exponentially starting from 0.1. If we cannot adhere to the budget in three iterations, the algorithm exits with the utility schedule at that point. We keep the number of iterations low because for each iteration recomputation we pay in terms of time and energy.

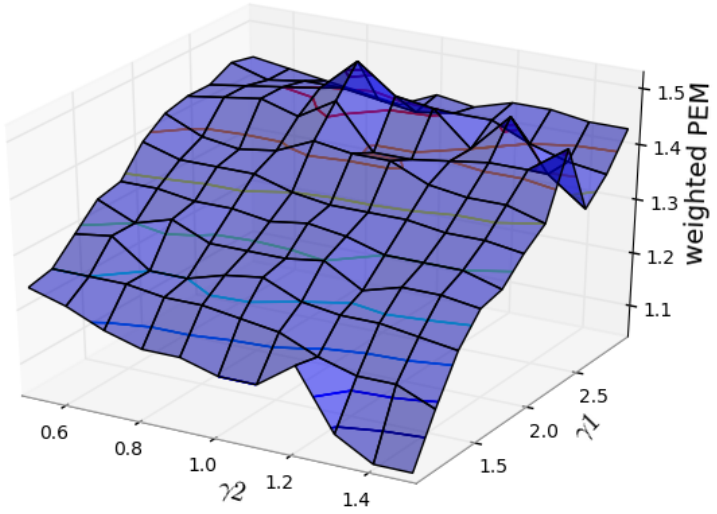

Figure 14: Effect of gamma values on $wPEM$.

7.3 Power envelope study

Figure 16 shows how the utility scheduler adapts to decreasing power envelopes. The maximum power achieved by a given group of applications may vary across our workload sets. Thus setting the same power budget for each one of them is unfair. Instead we need to find out what is the maximum power achievable by a set of four threads and use that for our analysis. Fortunately this is not difficult because highest power is consumed when the highest IPC thread runs on the fat core all the time. Thus we calculate this maximum power for each workload and then dial it down in

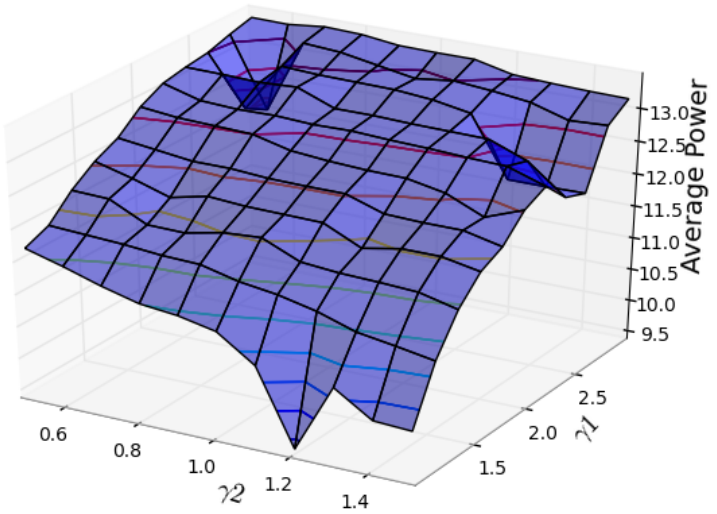

Figure 15: Effect of gamma values on Average Power.

steps of 10 percent. The figure plots the percentage overshoot error on the log scale as the budget is scaled down from 100% to 60% of maximum power. It should also be noted that in reality the budget will not be calculated this way, this is just for the sake of analysis so that we can stress test our algorithm.

The reason we evaluate a peak overshoot error is that our problem definition is not one of power tracking and we only care about adhering to a budget and finding a minima under it. At very low budgets many workloads do not have a feasible solution to the inequality constraint. In such cases adhering to power budget is not possible without resorting to complimentary techniques like switching of cores or voltage-frequency scaling. In such a scenario we can either decrease the frequency of the whole migration domain and start over or we can request the higher level thread schedulers(HLTS) for more power budget. This can also be used as feedback to HLTS for power demand of groups of threads.

Most workloads successfully adhere to budgets upto 75% of the maximum power budget. Below that more often than not the solution is not feasible and the utility scheduler gives the solution with lowest possible power. This shows that thread scheduling for power management technique roughly has a power budget modulation

window of 25%.

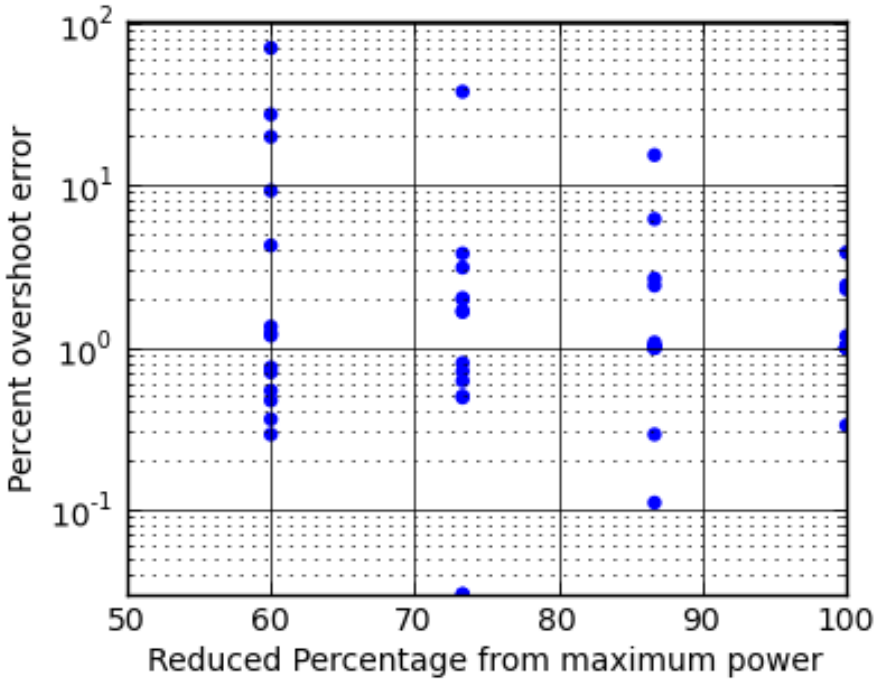

Figure 16: Percentage overshoot error with decreasing power envelopes.

Figure 17 shows the effect of decreasing power budget on $wPEM$. As expected the $wPEM$ decreases with decreasing power budget. But the decrease is different for different workloads. This is because the minima points for different workloads decides the power at that point. For some workloads the power is inherently much lower than the maximum power. They are less affected by reducing the power budget.

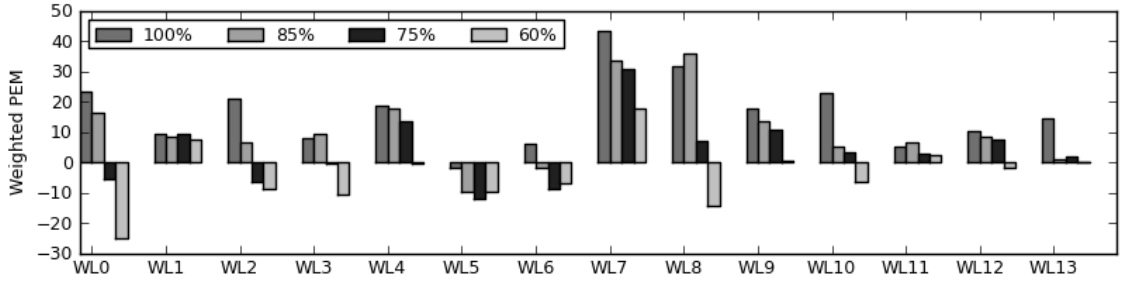

Figure 17: Weighted PEM for Quad core workloads as the power envelope is dialed down

7.4 Throughput Formulation

We now try to see how our scheduler fares when we formulate the cost objective as weighted throughput. Figure 18 shows the $wIPC$ impact of Utility Scheduling and COBYLA. As we can see there is no major qualitative difference between the two formulations. In this case the $wIPC$ has much higher improvement but the $wPEM$ has lesser gains than when we formulate the problem for ED^2. Also it should be noted that the $hIPC$ is not as degraded as the one for the ED^2 formulation. This is because of the squared term in the ED^2 formulation.

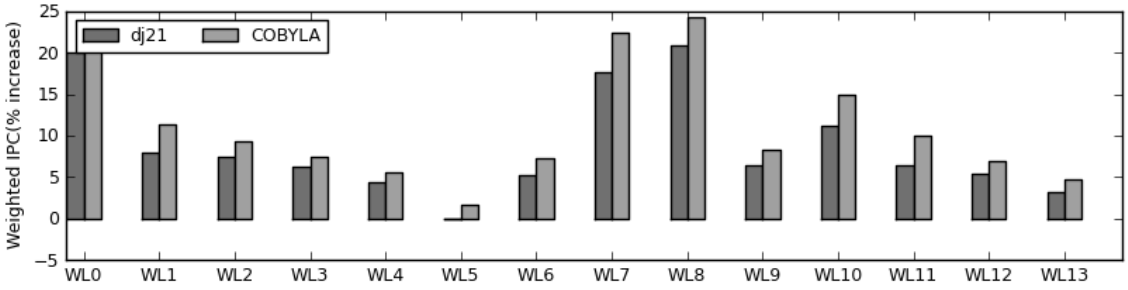

Figure 18: Weighted Throughput for Quad core workloads when optimized for Throughput

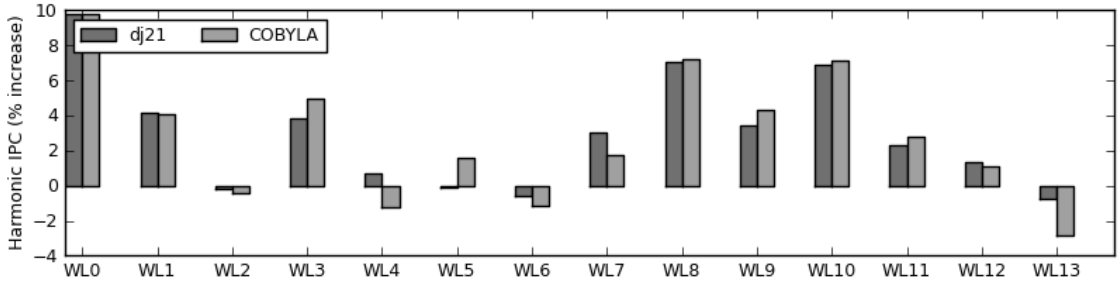

Figure 19: Harmonic Throughput for Quad core workloads when optimized for Throughput

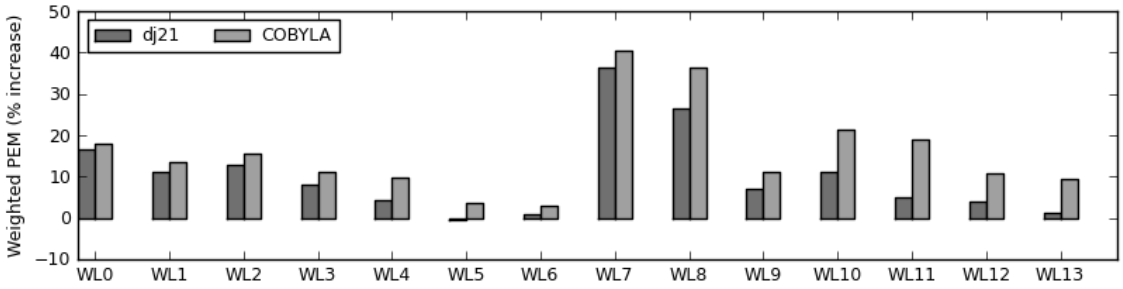

Figure 20: Weighted PEM for Quad core workloads when optimized for Throughput

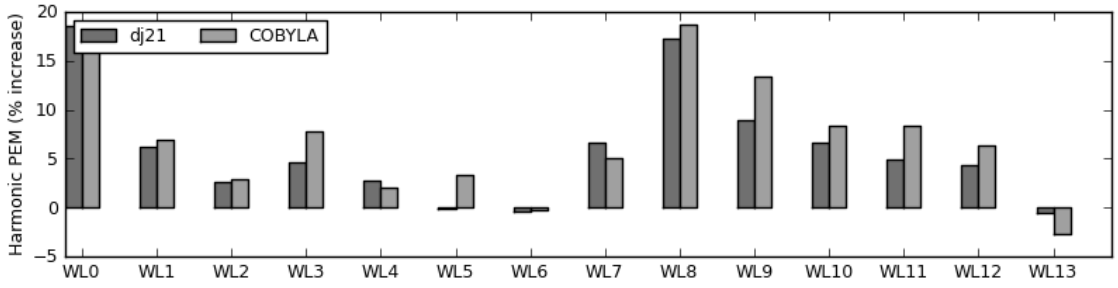

Figure 21: Harmonic PEM for Quad core workloads when optimized for Throughput

CHAPTER VIII

PREVIOUS WORK

There have been many proposals that have addressed thread scheduling for asymmetric processors with a few major techniques proposed for *micro-scheduling*. Teodorescu et al [13] formulated the DVFS problem as a linear programing technique and propose a variation-aware scheduler. Their technique cannot be used to define other objectives like ED^2. Krishna et al [12] proposed a fairness-aware throughput maximization algorithm. Their substrate is an SMP where asymmetry is due to operation of cores at different frequencies. They aimed to achieve uniform performance at a mean frequency inside an OS quantum by performing migrations on the order of million cycles. Our migration interval is relatively higher and is aimed at power management. Winter et al [14] use the Hungarian algorithm to solve a graph assignment formulation of the scheduling problem. The disadvantage of their algorithm is that it is computationally demanding.

Many past works have focused on *OS level scheduling* where application performance is sampled on different cores and the best assignment is run for a steady long phase. These schedules suffer from low reactivity and some of the problems mentioned in Chapter II. Kumar et al [5] were the first to propose asymmetric multiprocessors for lower power operation. Bower et al [1] elaborate on the various sources of asymmetry and the reasons why schedulers need to be aware of asymmetry. Sergey et al [16] have looked at contention aware thread scheduling at the OS level. Tongli et al [8] have proposed schedulers for shared ISA heterogeneous systems. Nagesh et al [6] proposed age based scheduling for equal completion times of threads. Most of these techniques handle migrations at a much higher level and thus it is difficult for them to adhere

to the lowering power budgets.

Cache Partitioning has been a major area of research for many years. Although there are numerous proposals for cache-partitioning, we mention the relevant and recent ones. Qureshi et al [11] suggested using cache utility monitors to allocate hard partitions in a cache. Our notion of thread utility is inspired by and is very similar to their notion of cache utility. Xie et al [15] proposed a simple extension to the utility cache partitioning scheme by defining new insertion policies that implicitly partition caches without making hard partitions. We use their notion of insertion priority to define our cache partitioning policy. Jaleel et al [3] suggested Thread Aware Dynamic Insertion Policy which is thrashing resistant by choosing to insert lines at the bottom of the stack or the top. All these techniques are targeted at symmetric multiprocessors. To the best of our knowledge there are no asymmetry aware cache partitioning techniques.

CHAPTER IX

CONCLUSION

This thesis has focused on techniques for maximizing the energy efficiency for asymmetric multicore architectures within a given power envelope. We have proposed a micro-scheduling approach and an associated optimization for generating thread micro-schedules. The micro-schedules are based on a notion of thread utility - intended to capture the affinity of a thread for execution on a specific core type. Practically, thread utility varies over time and as a function of the thread demand (e.g., compute phase vs. memory phase). Thus, the proposed micro-scheduling framework operates periodically every T cycles. Further, we observed that in an asymmetric architecture, the demand for cache resources are also asymmetric leading to non-uniform sharing of the cache the effects of which can feedback to the thread scheduler with negative consequences. Consequently we propose that the micro-scheduling step be coordinated with cache partitioning. We achieve this coordinated management by using the thread utility to drive the sharing of the cache between the asymmetric cores bu biasing the insertion policy as a function of thread utility.

Apart from describing the the advantages of micro-scheduling we would also like to point out some issues facing the application of such a scheme in a real system. Migration costs can increase exorbitantly when micro-scheduling is scaled to larger number of cores. Thus we envisage that such a micro-scheduling concept is going to be contained in small domains of 4 to 8 cores. It is unclear if increasing domain size more than that would have any incremental benefit with added increase in computation and migration costs.

We have considered a particular form of asymmetry where the cores are of different

types but qualitatively our framework and insights can be applied to many other forms of asymmetry such as voltage-frequency asymmetry or asymmetry due to process variations. In systems that have cores operating at different frequency and voltage, the weighted quadratic function can be applied and this application also suffers from the cache effects we have highlighted. Votage frequency changes can be applied to a system we have proposed as well where the frequency of fat or thin cores can be changed. The micro-scheduler in that scenario would take input values in terms of seconds rather than cycles.

REFERENCES

[1] Bower, F. A., Sorin, D. J., and Cox, L. P., "The impact of dynamically heterogeneous multicore processors on thread scheduling," *IEEE Micro*, vol. 28, pp. 17–25, May 2008.

[2] Hamilton, J., "Overall data center costs," *http://perspectives.mvdirona.com/2010/09/18/OverallDataCenterCosts.aspx*.

[3] Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely, Jr., S., and Emer, J., "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, (New York, NY, USA), pp. 208–219, ACM, 2008.

[4] Johnson, S. G., "The nlopt nonlinear-optimization package," *http://ab-initio.mit.edu/nlopt*, vol. 2, 2003.

[5] Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M., "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," pp. 81–92, 2003.

[6] Lakshminarayana, N. B., Lee, J., and Kim, H., "Age based scheduling for asymmetric multiprocessors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 25:1–25:12, ACM, 2009.

[7] Li, S., Ahn, J. H., Strong, R., Brockman, J., Tullsen, D., and Jouppi, N., "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 469 –480, Dec. 2009.

[8] Li, T., Brett, P., Knauerhase, R. C., Koufaty, D. A., Reddy, D., and Hahn, S., "Operating system support for overlapping-isa heterogeneous multi-core architectures," in *HPCA*, pp. 1–12, 2010.

[9] Loh, G. H., Subramaniam, S., and Xie, Y., "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *In Proc. of the Int. Symp. on Performance Analysis of Systems and Software*, 2009.

[10] MJD, P., "A direct search optimization method that models the objective and constraint functions by linear interpolation," *Advances in Optimization and Numerical Analysis, Kluwer Academic, Dordrecht*, 1994.

[11] Qureshi, M. K. and Patt, Y. N., "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, (Washington, DC, USA), pp. 423–432, IEEE Computer Society, 2006.

[12] Rangan, K. K., Powell, M. D., Wei, G.-Y., and Brooks, D., "Achieving uniform performance and maximizing throughput in the presence of heterogeneity," in *HPCA*, pp. 3–14, 2011.

[13] Teodorescu, R. and Torrellas, J., "Variation-aware application scheduling and power management for chip multiprocessors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 363–374, IEEE Computer Society, 2008.

[14] Winter, J. A., Albonesi, D. H., and Shoemaker, C. A., "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 29–40, ACM, 2010.

[15] Xie, Y. and Loh, G. H., "Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, (New York, NY, USA), pp. 174–183, ACM, 2009.

[16] Zhuravlev, S., Blagodurov, S., and Fedorova, A., "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, (New York, NY, USA), pp. 129–142, ACM, 2010.